TMA4215 Numerical Mathematics –Project 2

October 5, 2020

1 Practical information

The main objective of the project is for you to apply some of your knowledge from this course, in particular from modules 3 and 4, to train a neural network as specified.

Groups. You are encouraged to form groups of 1-3 students. The maximum allowed group size is 3. Feel free to use e.g. Piazza to find each other to form a group. Eventually it will be possible to register groups in Inspera. One person (per group) can create a group, and will get a PIN code which can given to the other group members for registering.

Requirements for submission. Submission in Inspera. Each group must submit their own report, it is not allowed to copy from other groups.

Deadline. November 2, 2020, 23:59. The deadline is absolute, no extensions can or will be given.

Supervision. Every Wednesday and Thursday, 16:00-17:30 via Piazza and Zoom. Questions can be asked any time on Piazza and will be answered as quickly as possible.

Peer review. After the deadline, each group will get the code of one of the other groups, and is required to write a short evaluation of the code. More information about what one should comment on will be given later in a separate document. This evaluation must be submitted in order to get a score for the group. The evaluation will not otherwise affect the point score of the group (the evaluating or evaluated one). But each evaluated group will get access to the evaluation which will be done anonymously. We ask all groups to give constructive feedback and avoid using harsh words.

2 Introduction

The traditional way of modelling and simulating physical phenomena, such as fluid flows, electromagnetism, chemical reaction dynamics, molecular dynamics etc, is to use physics to write down a mathematical model. Usually the model is in the form of partial or ordinary differential equations, possibly including parameters. If these parameters are

known to the user, all we need to do is to approximate the model by some numerical method to obtain the solution we are looking for. In other cases, the parameter values are unknown, and the purpose is to find values that make the model agree with measurements. These are called inverse problems, and they are somewhat related to what we shall be doing in this project. Here we shall be considering Hamiltonian systems and what we train a model to find the Hamiltonian or energy function based on data. The data consists of values of the Hamiltonian at a set of points in the phase space (solution space).

This project is based on ideas that have appeared in the literature very recently, see e.g. [2]. It has however been substantially simplified to make it suitable for a 3rd year project in Numerical mathematics.

3 Hamiltonian dynamics

We are considering models where the unknowns can be thought of as generalised positions and momenta, $y = (q, p)$. In the Hamiltonian framework, we have at our disposal a Hamiltonian, or energy function, $H : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ which is preserved along exact solutions. In fact, the differential equations can be expressed as follows

$$\dot{q} = \frac{\partial H}{\partial p}(p, q)$$
$$\dot{p} = -\frac{\partial H}{\partial q}(p, q)$$

where the dot above a symbol signifies differentiation with respect to time. The partial derivatives can be thought of as gradients with respect to each of q and p, e.g. $\frac{\partial H}{\partial p} = \nabla_p H$. Systems of this form are called *Hamiltonian systems*. They have two important properties, the first one is that $H(p, q)$ is constant along the exact solution, which is easy to see, since by the chain rule and the differential equations

$$\begin{aligned}\frac{d}{dt}H(q,p) &= \frac{\partial H}{\partial q}(q,p)\cdot\dot{q} + \frac{\partial H}{\partial p}(q,p)\cdot\dot{p} \\ &= \frac{\partial H}{\partial q}(q,p)\cdot\frac{\partial H}{\partial p}(q,p) + \frac{\partial H}{\partial p}(q,p)\cdot\left(-\frac{\partial H}{\partial q}(q,p)\right) = 0\end{aligned}$$

The other important property of the exact solution involves the Jacobian of the flow map (solution map). The flow of the vector field corresponding to the Hamiltonian H is denoted $\varphi_{t,H}(q_0, p_0)$. This is the map that, for any initial value (q_0, p_0) maps the solution to its value at time t, i.e. $\varphi_{t,H}(q_0, p_0) : (q_0, p_0) \mapsto (q(t), p(t))$.

For each t, this is a map from $\mathbb{R}^m \times \mathbb{R}^m$ into $\mathbb{R}^m \times \mathbb{R}^m$, and its Jacobian can be defined as a $2m \times 2m$-matrix

$$\Psi_t(q, p) = \frac{\partial \varphi_{t,H}}{\partial(q, p)}$$

On any open set of $\mathbb{R}^m \times \mathbb{R}^m$ it holds that

$$\Psi_t(q,p)^T J \Psi_t(q,p) = J, \qquad \text{with } J = \begin{bmatrix} 0 & I \\ -I & 0 \end{bmatrix} \tag{1}$$

We do not prove that relation, but note in passing that maps which have this property (such as $\varphi_{t,H}$) are called *symplectic*.

In mechanics, one often encounters a subclass of Hamiltonian functions corresponding to what is called *separable systems*. This means that the Hamiltonian has the particularly simple form $H(p,q) = T(p) + V(q)$ where the physical interpretation is that $T(p)$ is the kinetic energy and $V(q)$ is the potential energy of the system. Then the Hamiltonian system takes the simple form

$$\begin{aligned} \dot{q} &= \frac{\partial T}{\partial p}(p) \\ \dot{p} &= -\frac{\partial V}{\partial q}(q) \end{aligned} \tag{2}$$

where the physical interpretation is that $T(p)$ is the kinetic energy and $V(q)$ is the potential energy of the system. It should be noted that in practice, when the kinetic energy depends only on the momentum variable p, we usually have $T(p) = \frac{1}{2}p^T M^{-1} p$ where M is some diagonal, positive mass matrix. It is also common to rescale so that the mass disappears and one has simply $T(p) = \frac{1}{2}p^T p$.

One can, in fact, design numerical integrators (methods) which satisfy the symplecticity condition (1). For separable systems they can even be made explicit. The simplest example is the *symplectic Euler method* (see also Assignment 4) which reads

$$\begin{aligned} q_{n+1} &= q_n + h\frac{\partial T}{\partial p}(p_n) \\ p_{n+1} &= p_n - h\frac{\partial V}{\partial q}(q_{n+1}) \end{aligned} \tag{3}$$

3.1 Examples of separable Hamiltonian problems in mechanics

1. The nonlinear pendulum, q and p are scalars.

   $$H(p,q) = \frac{1}{2}p^2 + mg\ell(1 - \cos(\theta)$$

   where m is mass, g is gravitation constant and ℓ is length of pendulum.

2. Kepler two-body problem, q and p are both vectors in $\mathbb{R}^2$

   $$H(p,q) = \frac{1}{2}p^T p - \frac{1}{\sqrt{q_1^2 + q_2^2}}$$

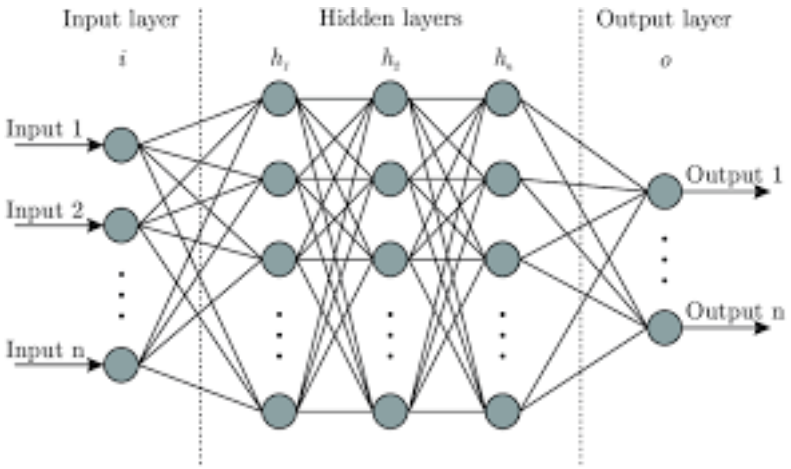
3. The Henon-Heiles problem. q and p are both vectors in $\mathbb{R}^2$

   $$H(p,q) = \frac{1}{2}p^T p + \frac{1}{2}q^T q + q_1^2 q_2 - \frac{1}{3}q_2^3$$

   This problem has chaotic behaviour in some parts of the phase space. See [1, p. 15–18] for more information. (Note that this reference is a book that should be available for NTNU students).

4 Artificial neural networks

Neural networks as a computational method goes back several decades, but has recently gained considerable interest within data science and machine learning in particular. We here give a brief introduction to a specific form of a neural network, just what we need in this project. Generally, one can think of neural networks as a collection of layers of information, where the first layer is the input data, then there is a number of hidden layers, often referred to as feature spaces, and finally an output layer. We can think of each layer of information to be an element of a linear space $\mathcal{Y}_k$ of some dimension d_k that may be different for each layer and where $\mathcal{Y}_0$ is the input layer. Our convention here will be to name the hidden layer spaces $\mathcal{Y}_1, \ldots, \mathcal{Y}_K$ and then add a final output



layer, $\mathcal{Y}_{K+1}$.

Between any two consecutive layers, there is a transformation between the two linear spaces, say $\Phi_k : \mathcal{Y}_k \to \mathcal{Y}_{k+1},\ k = 0, \ldots, K..$ Each such transformation has a set of parameters θ_k that determines the Φ_k, more concrete information about this follows later. We sometimes set $\theta = \{\theta_k\}_{k=0}^{K}$. Abstractly, we can think of the neural networks as a function approximant. Suppose that we are seeking some function $F : \mathcal{Y}_0 \to \mathcal{Y}_{K+1}$, that maps the input space to the output space, and which is known for a large set of data, i.e. $F(y_i) = c_i$ for a set $\{(y_i, c_i)\}_{i=1}^{I}$. In supervised learning, we typically try to find values of the parameters θ_k in each layer that in some sense minimises the differences $F(y_i) - c_i$ for all the data, we get back to a precise definition of this accumulated error later. Then, we can try to generalise, that is, to check how well the resulting $F(y)$ approximates new data.

The ResNet architecture. The structure of a neural network, including the definitions of Φ_k is called its architecture. One popular architecture is ResNet, first presented in [3], defined as follows: We let d_0 be the dimension of the input space $\mathcal{Y}_0$. Then we choose some integer $d \geq d_0$ which will be the dimension of all the hidden layers, in fact, here we can think of all the hidden layers as being the same linear space $\mathbb{R}^d$. In the

present application, the output layer space is $\mathbb{R}$ and has dimension 1, $d_{K+1} = 1$, since we approximate scalar valued functions.

$$\Phi_k(y) = y + h\sigma(W_k y + b_k), \quad k = 0, \ldots, K-1, \quad \Phi_K(y) = \eta(y^T w + \mu)$$

where $(W_k, b_k) =: \theta_k$, $0 \leq k \leq K-1$ are the parameters of layer k, where $W_k \in \mathbb{R}^{d\times d}, b_k \in \mathbb{R}^d$. In the very first layer, if $d_0 < d$, we first embed the input data in $\mathbb{R}^d$ and use the same mapping. In the last hidden layer, the parameters $\theta_K = (w, \mu)$ are used, where $w \in \mathbb{R}^d$ and $\mu \in \mathbb{R}$. σ is a given nonlinear *activation function*, a scalar function $\sigma : \mathbb{R} \to \mathbb{R}$ that acts element-wise when applied to vectors or matrices. Examples of popular activation functions are

$$\sigma(x) = \tanh x \text{ (sigmoid)} \quad \text{and} \quad \sigma(x) = \max\{0, x\} \text{ (ReLU)}$$

The function $\eta(x)$ appearing in the final layer is called the *hypothesis function*. In this particular application it plays a minor role and can be omitted. But we suggest in any case to include it in your model setup but allow for it to be the identity function $\eta(x) = x$. Another used option is

$$\eta(x) = \frac{1}{2}(1 + \tanh\frac{x}{2})$$

Note that in the present case, one should make sure that the range of the hypothesis function contains the range of the data values c_i. The best way to achieve this is perhaps to scale the data.

5 The model we shall use

Our objective is to estimate a separable Hamiltonian function $H(p, q) = T(p) + V(q)$. We do this by training one neural network for each of $T(p)$ and $V(q)$. We will use the ResNet architecture as neural network.

5.1 Input data

In a realistic setting, it would be relevant to give as input data discrete trajectories generated by the dynamics inferred by some separable Hamiltonian system. One could think of a collection trajectories of a system being measured to yield data of the form (t_i, q_i, p_i) for an observed Hamiltonian dynamical system. We could have set up a neural network for each of $T(p)$ and $V(q)$. Then we could derive from this network the gradients appearing in (2), and then computed the trajectories by a numerical method. This would give us model values for the discrete trajectories that could be compared to the given data, and we could seek values of the parameters θ of the network that would minimise the discrepancy between the model and the data. This is however a somewhat tedious process to define manually. There are software packages, such as `PyTorch` or `Tensorflow` that could resolve these technical difficulties with ease, but we prefer here to consider

a simpler problem and to do all the programming just with the `numpy` library. This is more relevant to this course.

So let us consider the case indicated earlier of approximating $F(y)$ where $F : \mathbb{R}^d \to \mathbb{R}$, and now keep in mind that $F(y)$ will later be either $T(p)$ or $V(q)$. In understanding the model and debugging your programme it is useful to apply example data that you generate yourself with some given function $F(y)$. In this project, data will also be provided where the source is unknown (no explicit function given).

Here are some suggestions of simple example functions you can use including there domains of definition

d_0	d	$F(y)$	Domain
1	2	$\frac{1}{2}y^2$	$[-2, 2]$
1	2	$1 - \cos y$	$[-\frac{\pi}{3}, \frac{\pi}{3}]$
2	4	$\frac{1}{2}(y_1^2 + y_2^2)$	$[-2, 2] \times [-2, 2]$
2	4	$-\frac{1}{\sqrt{y_1^2+y_2^2}}$	Exclude origin (0,0)

One way of generating data is to use a random generator for alle the inputs y_i, for instance uniformly over the domain if that is feasible. For each data point y_i you then compute $c_i = F(y_i)$. You must make your own choice for how many data points to use, remember that models improve when you increase the amount of data, but at the cost of increasing computation time.

Scaling your data. In training neural networks based on arbitrary data, it is always a good idea to scale both the input (for us y_i) and the output (here c_i). A linear transformation with shift and scaling is often sufficient. Since the architectures often make use of functions acting component-wise, it may be a good idea to applied a so called min-max transformation. For the data points you could search for bounds $a = \min_{i,j} y_{ij}$ and $b = \max_{i,j} y_{ij}$ where y_{ij} is component j of data point y_i. Then you could set

$$\tilde{y}_i = \frac{1}{b-a}((b - y_i)\alpha + (y_i - a)\beta)$$

which would cause all $\tilde{y}_i$ to have all its components in $[\alpha, \beta]$. A possible choice is $\alpha = 0, \beta = 1$. A similar type of scaling could be done for the c_i values.

5.2 Objective function

In what follows, we shall denote the neural network approximant to $F(y)$ by $\tilde{F}(y; \theta)$. We also define the intermediate values in each hidden layer by $z^{(k)}$. Note that we have the recursion for $k = 0, \ldots, K - 1$

$$z^{(k+1)} = \Phi_k(z^{(k)}) = z^{(k)} + h\sigma(W_k z^{(k)} + b_k), \quad z^{(0)} = y.$$

Finally, we then obtain $\tilde{F}(y; \theta) = \eta\left((z^{(K)})^T w + \mu\right)$.

We see that each input data value y_i is propagated through the network independently of each other. In fact, by defining a $d \times I$-matrix $Y = [y_1, \ldots, y_I]$ whose columns are the data values, we can also define intermediate values for all data simultaneously, setting $Z^{(k)} = [z^{(1)}, \ldots, z^{(I)}]$ and the recursion still makes sense, i.e.

$$Z^{(k+1)} = \Phi_k(Z^{(k)}) = Z^{(k)} + h\sigma(A_k Z^{(k)} + b_k), \quad Z^{(0)} = Y. \tag{4}$$

so we are now propagating $d \times I$-matrices rather than vectors. In the final layer, we set

$$\Upsilon = \tilde{F}(Y;\theta) = \eta\left((Z^{(K)})^T w + \mu \mathbf{1}\right) \tag{5}$$

which is a vector of function values,

$$\Upsilon = [\Upsilon_1, \ldots, \Upsilon_I] = [\tilde{F}(y_1;\theta), \ldots, \tilde{F}(y_I;\theta)]^T.$$

Letting $c = [c_i, \ldots, c_i]^T$ where $c_i = F(y_i)$ are the given data, we can define an objective function

$$J(\theta) = \frac{1}{2}\|\tilde{F}(Y;\theta) - c\|^2 = \frac{1}{2}\|\Upsilon - c\|^2 \tag{6}$$

5.3 The training process

The overall idea here is to search for values of the parameters θ that minimise the objective function (6). This is usually done by means of a gradient based optimisation method. The simplest possible scheme to use is the following: From an initial guess $\theta^{(0)}$ and a "learning parameter" τ, set

$$\theta^{(r+1)} = \theta^{(r)} - \tau \nabla_\theta J(\theta^{(r)}), \ r = 0, 1, \ldots \tag{7}$$

and the iteration is terminated when either a maximum number of iterations is done or some tolerance criterion is met. The crucial job here is to be able to obtain the gradient $\nabla_\theta J(\theta)$ for a given parameter set, then one can easily apply (7) or some other gradient based scheme, such as the Adam method (do not confuse with Adams linear multistep methods, which is something entirely different).

Computing the gradient. The parameter set known under the name θ is a collection of several parts in the ResNet model

$$\theta = \{W_0, \ldots, W_{K-1}, b_0, \ldots, b_{K-1}, w, \mu\}$$

Thus, $\nabla_\theta J(\theta)$ is made up of gradients with respect to each of the parts

$$\frac{\partial J}{\partial W_k}, \quad \frac{\partial J}{\partial b_k}, \quad \frac{\partial J}{\partial w}, \quad \frac{\partial J}{\partial \mu},$$

all evaluated in $\theta^{(k)}$. The calculation may look overwhelming at first, but essentially all we need is the chain rule. The easiest part is perhaps that of w and μ, looking at (6)

and (5) we identify the dependence of J on these two parameters. We find by the chain rule

$$\frac{\partial J}{\partial \mu} = \frac{\partial J}{\partial \Upsilon} \cdot \frac{\partial \Upsilon}{\partial \mu}$$

Computing these derivatives, we get

$$\frac{\partial J}{\partial \mu} = \eta'((Z^{(K)})^T w + \mu \mathbf{1})^T (\Upsilon - c), \tag{8}$$

a scalar quantity. Computation of $\frac{\partial J}{\partial w}$ follows a similar pattern, but remember that the derivative here is a vector of dimension d because w has d components that we differentiate with respect to. The end result is

$$\frac{\partial \mathcal{J}}{\partial w} = Z^{(K)} \left[(\Upsilon - c) \odot \eta'((Z^{(K)})^T w + \mu) \right] \tag{9}$$

where we have introduced the Hadamard product $\odot$ between two matrices (vectors) of the same dimension

$$(A \odot B)_{ij} = A_{ij} \cdot B_{ij}.$$

`numpy.multiply` can be used for this operation, but you can also simply use '*' (as opposed to '@'), test on a simple example. Note that $A \odot B = B \odot A$. If we differentiate J with respect to W_k or b_k for any $k \in \{0, \ldots, K-1\}$, the situation is more complicated since the chain rule must be used repeatedly. We present here just the final result.

$$P^{(K)} = \frac{\partial J}{\partial Z^{(K)}} = w \cdot [(\Upsilon - c) \odot \eta'((Z^{(K)})^T w + \mu \mathbf{1})]^T \tag{10}$$

$$P^{(k-1)} = P^{(k)} + h\, W_{k-1}^T \cdot [\sigma'(W_{k-1} Z^{(k-1)} + b_{k-1}) \odot P^{(k)}] \tag{11}$$

$$\frac{\partial J}{\partial W_k} = h\, [P^{(k+1)} \odot \sigma'(W_k Z^{(k)} + b_k)] \cdot (Z^{(k)})^T \tag{12}$$

$$\frac{\partial J}{\partial b_k} = h[P^{(k+1)} \odot \sigma'(W_k Z^{(k)} + b_k)] \cdot \mathbf{1} \tag{13}$$

A few clarifications. In this process you start by defining $P^{(K)}$ which is a $d \times I$-matrix. The dot after w in (10) is to be interpreted as an outer product, if $w \in \mathbb{R}^d$ and $u \in \mathbb{R}^I$ then wu^T is a $d \times I$-matrix with ij-element $w_i u_j$, it can be formed for instance using `numpy.outer`.

Next, notice also how the $Z^{(K)}$ computed from (4) feature in the expressions for $P^{(k)}$ they can be computed first by a forward sweep. Then they are used in (10) and (11) in the so called *back propagation* as one sweeps back from the final to the first layer. These same quantities are also used in (12) and (13).

A concise summary of the training algorithm. It is recommended that you structure the code in a more modular way than what is indicated by this algorithmic description.

Algorithm

Set number of layers K
Set learning parameter τ
Set h used on the transformations Φ_k
Set $\mathbf{Y}_0$ by reading or generating data
Set random initial values for weights W_k and b_k in all layers
while not converged
    for $k = 1 : K$
        Compute $Z^{(k)}$ from (4) and store in memory
    end
    Compute $P^{(K)}$ from (10) og store in memory
    Calculate the pieces of the gradient corresponding to the projection step (8), (9)
    for $k = K : -1 : 2$
        Compute $P^{(k-1)}$ from (11)
    end
    for $k = 0 : K - 1$
        Compute contributions to the gradient from (12) and (13)
    end
    Update W_k and b_k as indicated in (7) or from e.g. the Adam-method
end (while)

Note that in applying this algorithm to approximate Hamiltonians you need to do everything twice, both for $T(p)$ and $V(q)$.

The ADAM method. We here briefly summarise the ADAM method, an alternative to the standard gradient descent method (7) for optimisation. A useful webpage to check is
`https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`

Adam descent algorithm

$\beta_1 := 0.9$, $\beta_2 := 0.999$, $\alpha := 0.01$, $\epsilon := 10^{-8}$
$v_0 := 0$, $m_0 := 0$ (zero vectors)
for $j = 1, 2, \ldots$
    $g_j := \nabla_\theta J(\theta^{(j)})$
    $m_j := \beta_1 m_{j-1} + (1 - \beta_1) g_j$
    $v_j := \beta_2 v_{j-1} + (1 - \beta_2)(g_j \odot g_j)$
    $\hat{m}_j = \frac{m_j}{1-\beta_1^j}$
    $\hat{v} = \frac{v_j}{1-\beta_2^j}$
    $\theta^{(j+1)} := \theta^{(j)} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}$
end

Some remarks about notation used

- β_1^j means "β_1 i j'th power". (Same for β_2^j)
- $\sqrt{\hat{v}_j}$ is component-wise square root and ϵ is added to all components
- The division $\frac{\hat{m}_j}{\sqrt{\hat{v}_j}+\epsilon}$ is also component-wise

Stochastic gradient descent. In data science it is typical that the amount of available data can be enormous, and this can make every iteration of the optimization method very costly. There is a simple way of amending this problem called Stochastic gradient descent. Notice that every data point can be fed through the network independently of each other. Only in the objective function are they mixed. We could replace the full matrix appearing in (6) by a matrix with just a (small) selection or subset of the columns of $Y = [y_1, \ldots, y_I]$. The subset will typically consist of a fixed number, $\bar{I} \ll I$ columns or data points. The subset is changed from iteration to iteration and is chosen by drawing a random set of indices. This is drawing without replacement (norwegian: trekning uten tilbakelegging). One can sift through the entire data set one or several times in this way.

5.4 Testing

It is not necessarily a good idea to drive the residual as far as one possibly can towards zero. If there is little data compared to the number of parameters, it may also cause a phenomenon called overfitting, meaning that the model insists to work well also for outlier (inaccurate) data. The problem is of overfitting is usually cured if more data can be included in the training. Instead, it is important to see how the model generalises to new data, data that was not used in the training process. That is the real test.

In the testing phase, the parameters θ has already been determined from the training phase, and are fixed. This means that there is no use of back-propagation or optimisation anymore, we only do forward sweeps with (4).

6 Using the model to compute dynamics

We now revisit the separable Hamiltonian system in (2). Clearly one needs to compute the gradients appearing in the right hand side, i.e. $\frac{\partial T}{\partial p}$ and $-\frac{\partial V}{\partial q}$. We need to be able to do this from the neural network models. It is left as an exercise to you both to develop the gradient formulas via ResNet and to implement it in Python.

Once this has been achieved you should be able to implement numerical methods. The symplectic Euler method is already given in (3), but this is a method of order only one. A second order method is the Størmer-Verlet method partly attributed the to Norwegian mathematician and physicist Carl Størmer who used this method to do

calculations on particles in a magnetic field while studying the northern light. The method can be written as follows for the problem (2)

$$\begin{aligned} p_{n+\frac{1}{2}} &= p_n - \frac{\Delta t}{2}\frac{\partial V}{\partial q}(q_n) \\ q_{n+1} &= q_n + \Delta t \frac{\partial T}{\partial p}(p_{n+\frac{1}{2}}) \\ p_{n+1} &= p_{n+\frac{1}{2}} - \frac{\Delta t}{2}\frac{\partial V}{\partial q}(q_{n+1}) \end{aligned} \tag{14}$$

7 Requirements and suggestions for elements to include in the report

This project is supposed to be a little open, meaning that we do not provide a detailed specification of what you need to do for full score. All the conclusions you make should however be well documented through systematic numerical tests. Your code should be readable, well structured, and reasonably efficient. You should summarise what you have coded in the report, preferably with some comments about how and why it is structured the way it is. An object oriented implementation with high degree of generality and reusability is of course allowed, but it is not the main focus of this project.

We list here a few guidelines for elements that you may profit on doing or reporting on. You may fill on other points, and you may choose to skip some of the suggested points.

1. Implement functions for generating synthetic input data
2. Implement the neural network for training approximation of Hamiltonian function
   (a) Test the model by using the suggested functions
   (b) Investigate systematically what are optimal choices for K, τ, d, h and any other choices you need to make. Balance performance in the generalisation phase with time consumption of training.
   (c) Train the model for the case of data given (with unknown Hamiltonian function).
   (d) Try other alternatives for optimisation, such a the Adam method
   (e) Make use of convergence plots for getting an indication of the efficiency of your choices
   (f) Include an evaluation, mostly through experiments, on how well your trained model approximates the function you started with on the training data.
   (g) Do a similar evaluation on test data (that were not used in the training phase)
3. Derive the formulas for computing the gradient of the trained function ($\nabla_y F(y)$)

4. Implement these formulas for computing the gradient
5. Implement symplectic Euler and the Størmer-Verlet method for the Hamiltonian function
   (a) Try in particular to test it on the given Hamiltonians
   (b) Test to which extent the numerical solution preserves the Hamiltonian along trajectories
   (c) Then try it on the given data with unknown Hamiltonian

References

[1] E. Hairer, C. Lubich, and G. Wanner, *Geometric numerical integration: structure-preserving algorithms for ordinary differential equations*, vol. 31, Springer Science & Business Media, 2006.

[2] Zhengdao Chen and Jianyu Zhang and Martin Arjovsky and Léon Bottou, *Symplectic Recurrent Neural Networks*, arXiv 1909.13334, 2020.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition.* In IEEE Conference on Computer Vision and Pattern Recognition, pages 770–778, 2016.